Compiladores

Luís Fernando Cavalcante dos Santos

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brasil

luis.fernnando@gmail.com

Abstract. *Este documento têm como objetivo realizar uma descrição de todas as etapas envolvidas no desenvolvimento de um compilador para a linguagem TINY ++, etapas essas que são a análise léxica, sintática, semântica e geração de código.*

1. Introdução

Um compilador é basicamente um programa que traduz uma linguagem para outra. O programa recebe um código fonte em determinada linguagem e gera um código equivalente em uma linguagem alvo.

O processo de compilação pode ser dividido em quatro principais fases: Análise Léxica, Analise Sintática, Analise Semântica e por fim, a geração do código final.

O presente documento esta organizado da seguinte forma. A seção 2 especifica a linguagem. A seção 3 aborda a gramática, a seção 4 descreve sobre o analisador léxico, a seção 5 sobre a análise sintática e a seção 6 sobre a análise semântica.

2. Linguagem TINY ++

2.1. Especificações da Linguagem

Ainda pode ser definido:

- Número: 1 ou mais dígitos que podem ser inteiro ou flutuante (representação em notação científica ou não);
- Identificador: começa com uma letra e precede com N letras e números sem limite de tamanho;
- Comentários: cercados de chaves da seguinte forma: '...'
- É aceito somente uma variável global, e apenas antes de qualquer função.
- Apenas é possível declarar uma variável por linha.
- Uma função deve obrigatoriamente ser seguida de uma função Principal.
- É possível ter apenas uma única função, desde que seja a Principal.

A seguir a tabela com os tokens da linguagem

palavras reservadas	símbolos
se	+ soma
então	- subtração
senão	* multiplicação
fim	/ divisão
repita	= igualdade
flutuante	, vírgula
vazio	:= atribuicão
até	> maior
leia	(abre-par
escreve	(fecha-par
inteiro	: dois-pontos
retorna	
principal	

Figura 1. Tokens da Linguagem

2.2. Expressões Regulares e Autômato

Expressões Regulares representam os tokens utilizados pela análise léxica, assim, cada token é representado por uma expressão regular. Na tabela abaixo podemos ver as expressões utilizadas.

Tokens	Expressões
NUMERO	[0-9]+(.[0-9]+)?(e-?[0-9]+)?
ID	[a-zA-ZÀ-ÿ_][a-zA-ZÀ-ÿ0-9_]*
ESCRITA	".*"
COMENTARIO	({(.|\n)*?\})
NOVALINHA	\n+

Figura 2. Expressões Regulares da Linguagem

Tokens mais simples podem ser representados por autômatos finitos, como os representados nas tabelas abaixo.

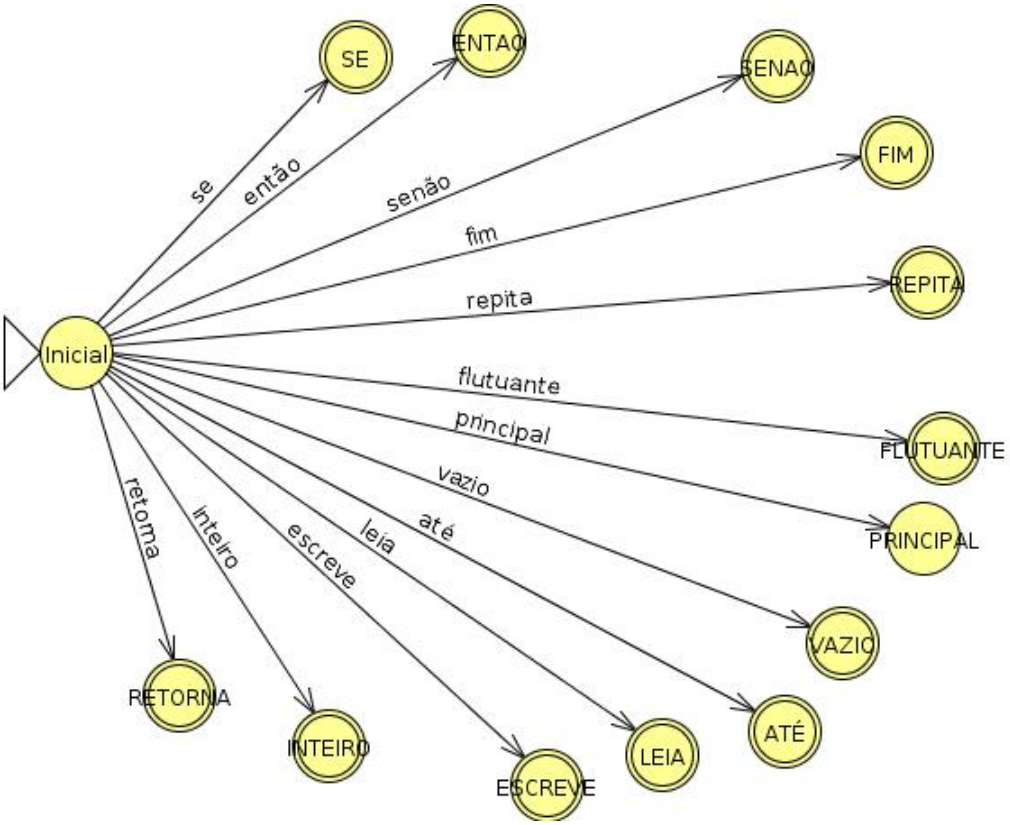

Figura 3. Reservadas

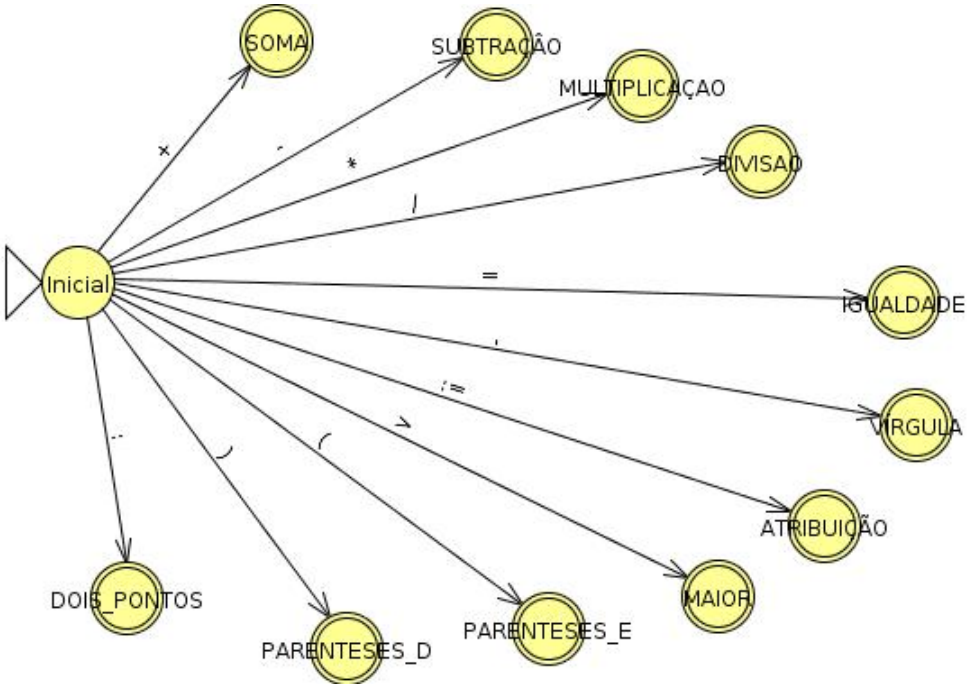

Figura 4. Simbolos

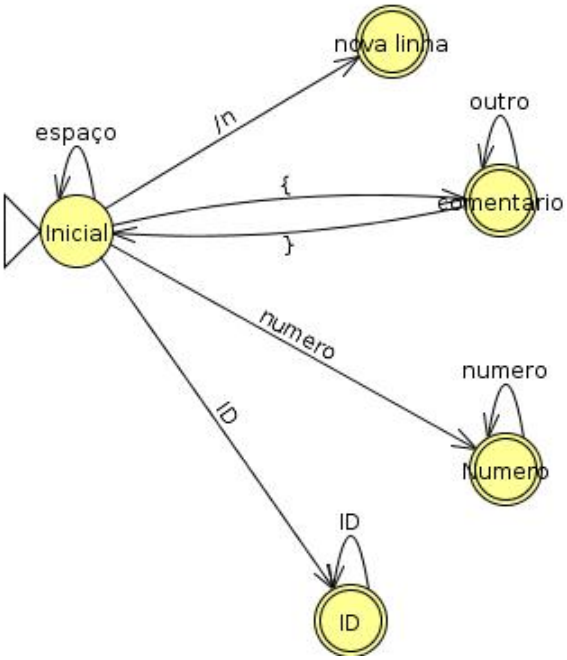

Figura 5. Outros

3. Gramática

A gramática na forma BNF para a linguagem Tiny ++ é apresentada a seguir:

```
<inicio> ::= programa
<programa> ::=  declaracao_tipo funcao_declaracao
<programa> ::=  funcao_declaracao
<funcao_declaracao> ::=  funcao funcao_declaracao
<funcao_declaracao> ::=  funcao_principal
<funcao> ::=  tipo_funcao id "(" parametros ")" expressoes "fim"
<funcao> ::=  tipo_funcao id "(" ")" expressoes "fim"
<funcao_principal> ::=  tipo_funcao PRINCIPAL "(" parametros ")" expressoes "fim"
<funcao_principal> ::=  tipo_funcao PRINCIPAL "(" ")" expressoes "fim"
<expressoes> ::=  expressoes expressao
<expressoes> ::=  expressao
<retorno> ::=  retorna factor
<tipo_funcao> ::=  inteiro
<tipo_funcao> ::=  flutuante
<tipo_funcao> ::=  vazio
<expressao> ::=  escreve_expre
<expressao> ::=  ler_expre
<expressao> ::=  atribuicao_expre
<expressao> ::=  repita_expre
<expressao> ::=  declaracao_tipo
<expressao> ::=  chama_func_expre
<expressao> ::=  se_expre
<expressao> ::=  retorno
<declaracao_tipo> ::=  inteiro ";"  id
<declaracao_tipo> ::=  flutuante ";"  id
<declaracao_tipo> ::=  vazio ";"  id
<atribuicao_expre> ::=  declaracao_tipo ":=" factor
<atribuicao_expre> ::=  identificador ":=" factor
<se_expre> ::=  SE condicao entao expressoes senao_expre "fim"
<senao_expre> ::=  senao expressoes
<senao_expre> ::=  <empty>
<escreve_expre> ::=  escreva "(" chama_func_expre ")"
<ler_expre> ::=  leia  "(" identificador ")"
<repita_expre> ::=  repita expressoes ATE condicao
<chama_func_expre> ::=  id "(" parametros_chamada ")"
<chama_func_expre> ::=  id "(" ")"
<parametros_chamada> ::=  factor VIRGULA parametros_chamada
<parametros_chamada> ::=  factor
<parametros> ::=  declaracao_tipo VIRGULA parametros
<parametros> ::=  declaracao_tipo
<factor> ::=  identificador
<factor> ::=  constante
<factor> ::=  "(" factor ")"
<factor> ::=  operacao_aritmetica
<identificador> ::=  id
<condicao> ::=  comparacao
<constante> ::=  NUMERO
<operador> ::=  "="
<operador> ::=  ">"
<operacao_aritmetica> ::=  factor "+" factor
<operacao_aritmetica> ::=  factor "-"  factor
<operacao_aritmetica> ::=  factor "*"  factor
<operacao_aritmetica> ::=  factor "/"  factor
<comparacao> ::=  factor operador factor
<comparacao> ::=  "(" comparacao ")"
```

4. Analisador Léxico

O analisador léxico separa em entidades ou tokens, a sequência de caracteres que representa o programa fonte. Um token consiste de um par ordenado (valor, classe). A classe indica a natureza da informação contida em valor. Outras funções do analisador léxico são, ignorar espaços em branco, comentários, e detectar erros léxicos.

Durante a análise léxica, os tokens são classificados como palavras reservadas, identificadores, símbolos especiais, constantes de tipos básicos (inteiro real, literal, etc.), entre outras categorias. O montador discutido neste relatório tem as seguintes especificações técnicas:

A seguir o código do Analisador Léxico:

Cdigo 1. Arquivo lex.py

```
1
2  import sys
3  import ply.lex as lex
4
5
6  class MyLexer(object):
7
8      #def __init__(self):
9
10     reservadas = {
11         'se': 'SE',
12         'então': 'ENTAO',
13         'senão': 'SENAO',
14         'fim': 'FIM',
15         'repita': 'REPITA',
16         'até': 'ATE',
17         'leia': 'LEIA',
18         'escreva': 'ESCREVA',
19         'inteiro': 'INTEIRO',
20         'flutuante': 'FLUTUANTE',
21         'vazio': 'VAZIO',
22         'principal': 'PRINCIPAL',
23         'retorna': 'RETORNA'
24     }
25
26     tokens = [
27         'NUMERO',
28         'SOMA',
29         'SUBTRACAO',
30         'MULTIPLICACAO',
31         'DIVISAO',
32         'IGUAL',
33         'MAIOR',
34         'VIRGULA',
35         'ATRIBUICAO',
36         'DOIS_PONTOS',
37         'PARENTESES_E',
38         'PARENTESES_D',
39         'ID'
40     ] + list(reservadas.values())
41
42     t_SOMA    = r'\+'
43     t_SUBTRACAO   = r'-'
44     t_MULTIPLICACAO   = r'\*'
45     t_DIVISAO  = r'/'
46     t_PARENTESES_E  = r'\('
47     t_PARENTESES_D  = r'\)'
48     t_IGUAL = r'='
49     t_MAIOR = r'>'
50     t_ATRIBUICAO = r':='
51     t_DOIS_PONTOS = r':'
52     t_VIRGULA = r','
```

```
53
54      def t_NUMERO(self, t):
55          r'[0-9]+(.[0-9]+)?(e-?[0-9]+)?'
56          if ('e' in t.value) or ('.' in t.value):
57              t.value = float(t.value)
58          else:
59              t.value = int(t.value)
60          return t
61
62      def t_COMENTARIO(self, t):
63          r'({(.|\n)*?\})'
64          t.lineno += t.value.count('\n')
65
66      def t_NOVALINHA(self, t):
67          r'\n+'
68          t.lexer.lineno += len(t.value)
69
70      def t_ID(self, t):
71          r'[a-zA-Z  - _ ][a-zA-Z  - 0 -9_]*'
72          t.type = self.reservadas.get(t.value, 'ID')
73          return t
74
75      t_ignore  = ' \t'
76
77      def t_error(self, t):
78          print("Illegal character %s" % t.value[0])
79          t.lexer.skip(1)
80
81      # Build the lexer
82      def build(self,**kwargs):
83          self.lexer = lex.lex(module=self, **kwargs)
84
85      # Test it outputhn
86      def test(self, data):
87          self.lexer.input(data)
88          while True:
89              tok = self.lexer.token()
90              if not tok:
91                  break
92              print (tok)
93
94  if (__name__ == "__main__"):
95      if (len(sys.argv) == 2):
96          f = open(sys.argv[1], 'r')
97          MLex = MyLexer()
98          MLex.build()
99          MLex.test(f.read())
100     else:
101         print("Error. Ex: lex.py script.tpp")
```

5. O Analisador Sintático

O analisador sintático tem como finalidade agrupar tokens que são gerados pelo analisador léxico em estruturas sintáticas, e assim é construindo uma árvore sintática de acordo com os tokens fornecidos.

Para a implementação desta seção, foi utilizada a ferramenta ply (YACC), o principal arquivo para a implementação é o parser.py, nele, todas as suas produções e ações são definidas, e, a partir disto, a ferramenta gera os arquivos parsetab.py e parser.out A

seguir será apresentado um trecho do código parser.py:

Cdigo 2. Arquivo parse.py

```
import sys
import ply.yacc as yacc

import AST
import lex

lexer = lex.MyLexer()
lexer.build()
tokens = lexer.tokens
has_error = False

precedence = (
    ('left', 'SOMA', 'SUBTRACAO'),
    ('left', 'MULTIPLICACAO', 'DIVISAO')
)

def p_programa(p):
    '''programa : declaracao_tipo funcao_declaracao
                | funcao_declaracao
    '''
    if(len(p) == 3):
        p[0] = AST.Node("programa",[p[1],p[2]])
    else:
        p[0] = AST.Node("programa",[p[1]])


def p_funcao_declaracao(p):
    '''funcao_declaracao : funcao funcao_declaracao
                | funcao_principal
    '''
    if(len(p) == 3):
        p[0] = AST.Node("funcao_declaracao",[p[1], p[2]])
    else:
        p[0] = AST.Node("funcao_declaracao",[p[1]])

def p_funcao(p):
    '''funcao : tipo_funcao ID PARENTESES_E parametros PARENTESES_D
       expressoes FIM
                | tipo_funcao ID PARENTESES_E PARENTESES_D
                  expressoes FIM
    '''
    if(len(p) == 8):
        p[0] = AST.Node("FUNCAO",[p[4],p[6]],[p[1],p[2]])
    else:
        p[0] = AST.Node("FUNCAO",[p[5]],p[2])

def p_funcao_principal(p):
    '''funcao_principal : tipo_funcao PRINCIPAL PARENTESES_E parametros
        PARENTESES_D expressoes FIM
                | tipo_funcao PRINCIPAL PARENTESES_E PARENTESES_D
                  expressoes FIM
    '''
    if(len(p) == 8):
        p[0] = AST.Node("FUNC_PRINCIPAL",[p[4],p[6]],[p[1],p[2]])
    else:
        p[0] = AST.Node("FUNC_PRINCIPAL",[p[5]],p[2])

def p_expressoes(p):
```

```
55      '''expressoes : expressoes expressao
56                    | expressao
57      '''
58      if(len(p) == 3):
59          p[0] = AST.Node("expressoes",[p[1], p[2]])
60      else:
61          p[0] = AST.Node("expressoes",[p[1]])
62
63  def p_retorno(p):
64      '''retorno : RETORNA factor'''
65      p[0] = AST.Node("retorno",[p[2]])
66
67  def p_tipo_funcao(p):
68      '''tipo_funcao : INTEIRO
69                     | FLUTUANTE
70                     | VAZIO
71      '''
72      if(p[1] == 'inteiro'):
73          p[0] = AST.Node("INTEIRO_FUNC",[])
74      elif(p[1] == 'flutuante'):
75          p[0] = AST.Node("FLUTUANTE_FUNC",[])
76      else:
77          p[0] = AST.Node("VAZIO_FUNC",[])
78
79
80  def p_expressao(p):
81      '''expressao : escreve_expre
82                   | ler_expre
83                   | atribuicao_expre
84                   | repita_expre
85                   | declaracao_tipo
86                   | chama_func_expre
87                   | se_expre
88                   | retorno
89      '''
90      p[0] = AST.Node("expressao",[p[1]])
91
92  def p_declaracao_tipo(p):
93      '''declaracao_tipo : INTEIRO DOIS_PONTOS ID
94                         | FLUTUANTE DOIS_PONTOS ID
95                         | VAZIO DOIS_PONTOS ID
96      '''
97      if(p[1] == 'inteiro'):
98          p[0] = AST.Node("INTEIRO",[],p[3])
99      elif(p[1] == 'flutuante'):
100         p[0] = AST.Node("FLUTUANTE",[],p[3])
101     else:
102         p[0] = AST.Node("VAZIO",[],p[3])
103
104 def p_atribuicao_expre(p):
105     '''atribuicao_expre : declaracao_tipo ATRIBUICAO factor
106                         | identificador ATRIBUICAO factor
107     '''
108     p[0] = AST.Node("ATRIBUICAO", [p[1],p[3]])
109
110
111 def p_se_expre(p):
112     '''se_expre : SE condicao ENTAO expressoes senao_expre FIM'''
113     p[0] = AST.Node("SE",[p[2],p[4],p[5]])
114
115
```

```
116 def p_senao_expre(p):
117     '''senao_expre : SENAO expressoes
118                    |
119     '''
120     if (len(p) == 3):
121         p[0] = AST.Node("SENAO",[p[2]])
122     else:
123         p[0] = AST.Node("SENAO",[])
124 
125 def p_escreve_expre(p):
126     '''escreve_expre : ESCREVA PARENTESES_E chama_func_expre
        PARENTESES_D
127     '''
128     p[0] = AST.Node("ESCREVA", [p[3]])
129 
130 def p_ler_expre(p):
131     '''ler_expre : LEIA PARENTESES_E identificador PARENTESES_D'''
132     p[0] = AST.Node("LEIA", [p[3]])
133 
134 
135 def p_repita_expre(p):
136     '''repita_expre : REPITA expressoes ATE condicao'''
137     p[0] = AST.Node("REPITA", [p[2],p[4]])
138 
139 def p_chama_func_expre(p):
140     '''chama_func_expre : ID PARENTESES_E parametros_chamada
        PARENTESES_D
141                         | ID PARENTESES_E PARENTESES_D
142     '''
143     if(len(p) == 5):
144         p[0] = AST.Node("chama_funcao",[p[3]],p[1])
145     else:
146         p[0] = AST.Node("chama_funcao",[],p[1])
147 
148 def p_exp_parametros_chamada(p):
149     '''parametros_chamada : factor VIRGULA parametros_chamada
150                           | factor
151     '''
152     if(len(p) == 4):
153         p[0] = AST.Node("parametros_chamada",[p[1],p[3]])
154     else:
155         p[0] = AST.Node("parametros_chamada",[p[1]])
156 
157 
158 def p_parametros(p):
159     '''parametros : declaracao_tipo VIRGULA parametros
160                   | declaracao_tipo
161     '''
162     if(len(p) == 4):
163         p[0] = AST.Node("parametros",[p[1],p[3]])
164     else:
165         p[0] = AST.Node("parametros",[p[1]])
166 
167 
168 def p_factor(p):
169     '''factor : identificador
170               | constante
171               | PARENTESES_E factor PARENTESES_D
172               | operacao_aritmetica
173     '''
174     if (len(p) == 4):
```

```
175         p[0] = AST.Node("factor",[p[2]])
176     else:
177         p[0] = AST.Node("factor",[p[1]])
178 
179 def p_identificador(p):
180     '''identificador : ID
181     '''
182     p[0] = AST.Node("ID",[],p[1])
183 
184 def p_condicao(p):
185     '''condicao : comparacao'''
186     p[0] = AST.Node("condicao",[p[1]])
187 
188 def p_constante(p):
189     '''constante : NUMERO'''
190     p[0] = AST.Node("constante",[],p[1])
191 
192 def p_operador(p):
193     '''operador : IGUAL
194                 | MAIOR'''
195     p[0] = AST.Node("operador",[],p[1])
196 
197 def p_operacao_aritmetica(p):
198     '''operacao_aritmetica : factor SOMA factor
199                 | factor SUBTRACAO factor
200                 | factor MULTIPLICACAO factor
201                 | factor DIVISAO factor'''
202     p[0] = AST.Node("operacao_aritmetica",[p[1],p[3]], p[2])
203 
204 def p_comparacao(p):
205     '''comparacao : factor operador factor
206                 | PARENTESES_E comparacao PARENTESES_D '''
207     if (p[1] == "("):
208         p[0] = AST.Node("comparacao",[p[2]])
209     else:
210         p[0] = AST.Node("comparacao",[p[1], p[2], p[3]])
211 
212 def p_error(p):
213     print("Erro sintatico " + str(p))
214     #print("Item ilegal: '%s', linha %d, coluna %d" % (p.value,p.lineno
        , p.lexpos))
215     global has_error
216     has_error = True
217 
218 # def t_NEWLINE(p):
219 #         r'\n+'
220 #         p.lineno += len(p.value)
221 
222 def ccparse(texto):
223     return parser.parse(texto)
224 
225 parser = yacc.yacc()
```

6. O Analisador Semântico

A análise semântica trata os aspectos sensíveis ao contexto da sintaxe das linguagens de programação. Por exemplo, uma regra como ”Todo identificador deve ser declarado antes de ser usado.”é verificada na análise semântica.

Para o desenvolvimento da análise semântica neste projeto fez-se a construção da tabela de símbolos, a qual é representada pela *struct* a seguir.

Cdigo 3. Arquivo parse.py

```
class table:
    def __init__(self, type, scope, value, numberParam=[],initialized=
        False, use = False):
         self.type = type
         self.scope = scope
         self.value = value
         self.initialized = initialized
         self.numberParam = []
         self.used = use
    def __repr__(self):
         value = len(self.numberParam)
         if (self.type == "FUNCAO" or self.type == "FUNC_PRINCIPAL"):
             return '(%s, %i)\n' % (self.type, value)
         else:
             return '(%s, %s,)\n' % (self.type, self.scope)
```

Os seguintes dados obrigatórios no armazenamento da tabela de símbolos são considerados:

- Identificador de procedimento: nome e quantidade de parâmetros formais
- Identificador de variáveis locais e globais: nome, tipo e escopo.

O analisador semântico considera as seguintes regras.

- Warnings deverão ser mostrados quando uma variável for declarada mais de uma vez;
- Uma variável não declarada. Lembrando que uma variável pode ser declarada: No escopo do procedimento (como expressão ou como parâmetro formal); No escopo global
- Tipo vazio não é compatível com nenhum outro tipo (erro). Variáveis são considerados na análise sintática
- A quantidade de parâmetros reais de uma chamada de procedimento deve ser igual a quantidade de parâmetros formais da sua definição;
- Warnings deverão ser mostrados quando ocorrer uma coerção implícita de tipos;
- Uma variável não inicializada;

7. Geração de Código

Para a geração de código foi utilizada a ferramenta llvm-lite (Low Level Virtual Machine). Inicialmente é gerado um código intermediário (IR), aceito pela llvm. Que possibilita a geração do código em (Assembly). É necessário realizar a importação de uma biblioteca externa implementada na linguagem C para efetuar as operações de entrada e saída que é ligada junto ao código gerado pelo compilador T++. A llvm possui várias otimizações a serem aplicadas, porém nenhuma foi utilizada.